## Objective 1: Setting up environment

Go to this website: <https://github.com/jafingerhut/p4-guide/blob/master/bin/README-install-troubleshooting.md>

Download any development VM image .ova file

Import this file into VirtualBox

Now log into the p4 user with password “p4”

You will now see this home screen:

A screenshot of a computer

AI-generated content may be incorrect.

Now clone the repository: <https://github.com/p4lang/tutorials>and start working!

## Objective 2: P4 programming

### Basic

First, fill in the code for the TODOs.

Code:

    action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port) {

        /\*

            Action function for forwarding IPv4 packets.

            This function is responsible for forwarding IPv4 packets to the specified

            destination MAC address and egress port.

            Parameters:

            - dstAddr: Destination MAC address of the packet.

            - port: Egress port where the packet should be forwarded.

            TODO: Implement the logic for forwarding the IPv4 packet based on the

            destination MAC address and egress port.

        \*/

        standard\_metadata.egress\_spec = port;

        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;

        hdr.ethernet.dstAddr = dstAddr;

        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;

   }

apply {

/\* TODO: fix ingress control logic

\* - ipv4\_lpm should be applied only when IPv4 header is valid

\*/

if (hdr.ipv4.isValid()) {

ipv4\_lpm.apply();

}

}

    action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port) {

        /\*

            Action function for forwarding IPv4 packets.

            This function is responsible for forwarding IPv4 packets to the specified

            destination MAC address and egress port.

            Parameters:

            - dstAddr: Destination MAC address of the packet.

            - port: Egress port where the packet should be forwarded.

            TODO: Implement the logic for forwarding the IPv4 packet based on the

            destination MAC address and egress port.

        \*/

        standard\_metadata.egress\_spec = port;

        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;

        hdr.ethernet.dstAddr = dstAddr;

        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;

   }

Run make run

Do a pingall to test basic connectivity

A black screen with white text

AI-generated content may be incorrect.

How is it different from Traditional/OpenFlow Networking concepts?

Here for the objective, we implement forwarding based on dest. MAC and egress port. Here instead of a traditional routing protocol, we are forwarding based on MAC address here. We also make the IPv4 header valid in the first place. We can then construct the packet with both ethernet and IP to send it off.

### Basic\_tunnel

First, fill in the code for the TODOs

Code:

// TODO: declare a new action: myTunnel\_forward(egressSpec\_t port)

    action myTunnel\_forward(egressSpec\_t port) {

        standard\_metadata.egress\_spec = port;

    }

    // TODO: declare a new table: myTunnel\_exact

    // TODO: also remember to add table entries!

    table myTunnel\_exact {

        key = {

            hdr.myTunnel.dst\_id: exact;

        }

        actions = {

            myTunnel\_forward;

            drop;

            NoAction;

        }

        size = 512;

        default\_action = drop();

    }

    apply {

        // TODO: Update control flow

        if (hdr.myTunnel.isValid()) {

            myTunnel\_exact.apply();

        } else if (hdr.ipv4.isValid()) {

            ipv4\_lpm.apply();

        }

    }

// TODO: emit myTunnel header as well

        packet.emit(hdr.myTunnel);

// TODO: Update the parser to parse the myTunnel header as well

    state parse\_myTunnel {

        packet.extract(hdr.myTunnel);

        transition parse\_ipv4;

    }

Now on start xterm on h1 and h2.

Do a ./receive.py on h2 and send a message using ./send.py

Initially, we should see the packet on h2. Now lets test with tunneling using the flag “—dst\_id 2”.

We should now see a new header that shows as “My Tunnel” with pid and dst\_id. This also works if the send is done for another ip, it will still be sent to h2.

Here’s an example of a send with no tunnel and with tunneling:  
A screenshot of a computer screen

AI-generated content may be incorrect.

How is it different from Traditional/OpenFlow Networking concepts?

This is different from these concepts because we are adding a new header into the packet that is defined by what we want. With other tunneling concepts, usually the actual information of the protocol is abstracted nested within a TCP header.

### Calc

First fill in the code TODOs

Code:

action send\_back(bit<32> result) {

        /\* TODO

         \* - put the result back in hdr.p4calc.res

         \* - swap MAC addresses in hdr.ethernet.dstAddr and

         \*   hdr.ethernet.srcAddr using a temp variable

         \* - Send the packet back to the port it came from

             by saving standard\_metadata.ingress\_port into

             standard\_metadata.egress\_spec

         \*/

        hdr.p4calc.res = result;

        bit<48> temp\_mac = hdr.ethernet.srcAddr;

        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;

        hdr.ethernet.dstAddr = temp\_mac;

        standard\_metadata.egress\_spec = standard\_metadata.ingress\_port;

    }

    action operation\_add() {

        send\_back(hdr.p4calc.operand\_a + hdr.p4calc.operand\_b);

    }

    action operation\_sub() {

        send\_back(hdr.p4calc.operand\_a - hdr.p4calc.operand\_b);

    }

    action operation\_and() {

        send\_back(hdr.p4calc.operand\_a & hdr.p4calc.operand\_b);

    }

    action operation\_or() {

        send\_back(hdr.p4calc.operand\_a | hdr.p4calc.operand\_b);

    }

    action operation\_xor() {

        send\_back(hdr.p4calc.operand\_a ^ hdr.p4calc.operand\_b);

    }

All we have to do here is “make” and then run “h1 python3 calc.py”

Here we will have an application that can do basic math, here is the ouput:

A computer screen with white text

AI-generated content may be incorrect.

How is it different from Traditional/OpenFlow Networking concepts?

It is different from these concepts because we are creating a new a header that’s has never been implemented before. The whole packet is custom and can do whatever operations we see fit. This is more defined than SDN because we are basically creating a new type of packet and defining how it works based off input and output, which differs heavily from just controlling already created protocols.

### ECN

First fill in the code TODOs

Code:

/\*

 \* TODO: split tos to two fields 6 bit diffserv and 2 bit ecn

 \*/

header ipv4\_t {

    bit<4>    version;

    bit<4>    ihl;

    bit<6>    diffserv;

    bit<2>    ecn;

    bit<16>   totalLen;

    bit<16>   identification;

    bit<3>    flags;

    bit<13>   fragOffset;

    bit<8>    ttl;

    bit<8>    protocol;

    bit<16>   hdrChecksum;

    ip4Addr\_t srcAddr;

    ip4Addr\_t dstAddr;

}

        /\*

         \* TODO:

         \* - if ecn is 1 or 2

         \*   - compare standard\_metadata.enq\_qdepth with threshold

         \*     and set hdr.ipv4.ecn to 3 if larger

         \*/

        if (hdr.ipv4.ecn == 1 || hdr.ipv4.ecn == 2) {

            if (standard\_metadata.enq\_qdepth > ECN\_THRESHOLD) {

                hdr.ipv4.ecn = 3;

            }

        }

To emulate this, run ./receive.py > h2.log on h2, iper -s -u on h22, ./send.py 10.0.2.2 “P4 is cool” 30 on h1, and iperf -c 10.0.2.22 -t 15 -u on h11

Then when we grep for tos h2.log we will see the tos values increase.

A screenshot of a computer

AI-generated content may be incorrect.

How is it different from Traditional/OpenFlow Networking concepts?

This is different from these concepts because typically packets will be dropped if there is too much congestion, but here it is not dropped and controlled via the notification system that was set up.

### Load\_balance

First fill in the code TODOs

Code:

   action set\_ecmp\_select(bit<16> ecmp\_base, bit<32> ecmp\_count) {

        hash(meta.ecmp\_select,

            HashAlgorithm.crc16,

            ecmp\_base,

            { hdr.ipv4.srcAddr,

              hdr.ipv4.dstAddr,

              hdr.ipv4.protocol,

              hdr.tcp.srcPort,

              hdr.tcp.dstPort },

            ecmp\_count);

    }

   apply {

        /\* TODO: apply ecmp\_group table and ecmp\_nhop table if IPv4 header is

         \* valid and TTL hasn't reached zero

         \*/

        if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 0) {

            ecmp\_group.apply();

            ecmp\_nhop.apply();

        } else {

            drop();

A screenshot of a computer screen

AI-generated content may be incorrect.

Here we run ./receive.py on h2 and h3 and run ./send.py 10.0.0.1 “P4 is cool” on h1. When we repeateldly send ./send.py, we will see packets received both on h2 and h3 in a randomly picked hash via the switch.

How is it different from Traditional/OpenFlow Networking concepts?

This has been and can be done in both concepts, but we see here we are implementing based on a computed hash that is calculated and then sent to a host based on the calculation. We see here the load balancing is a bit more in depth and is actually computed directly in the switch pipeline.

### MRI

First fill in the code TODOs

Code:

state parse\_ipv4\_option {

        /\*

        \* TODO: Add logic to:

        \* - Extract the ipv4\_option header.

        \*   - If value is equal to IPV4\_OPTION\_MRI, transition to parse\_mri.

        \*   - Otherwise, accept.

        \*/

        packet.extract(hdr.ipv4\_option);

        transition select(hdr.ipv4\_option.option) {

            IPV4\_OPTION\_MRI: parse\_mri;

            default: accept;

        }

   }

    state parse\_mri {

        /\*

        \* TODO: Add logic to:

        \* - Extract hdr.mri.

        \* - Set meta.parser\_metadata.remaining to hdr.mri.count

        \* - Select on the value of meta.parser\_metadata.remaining

        \*   - If the value is equal to 0, accept.

        \*   - Otherwise, transition to parse\_swtrace.

        \*/

        packet.extract(hdr.mri);

        meta.parser\_metadata.remaining = hdr.mri.count;

        transition select(meta.parser\_metadata.remaining) {

            0: accept;

            default: parse\_swtrace;

        }

    }

    state parse\_swtrace {

        /\*

        \* TODO: Add logic to:

        \* - Extract hdr.swtraces.next.

        \* - Decrement meta.parser\_metadata.remaining by 1

        \* - Select on the value of meta.parser\_metadata.remaining

        \*   - If the value is equal to 0, accept.

        \*   - Otherwise, transition to parse\_swtrace.

        \*/

        packet.extract(hdr.swtraces[MAX\_HOPS - meta.parser\_metadata.remaining]);

        meta.parser\_metadata.remaining = meta.parser\_metadata.remaining - 1;

        transition select(meta.parser\_metadata.remaining) {

            0: accept;

            default: parse\_swtrace;

        }

A screenshot of a computer program

AI-generated content may be incorrect.

To emulate this, run ./receive.py > h2.log on h2, iper -s -u on h22, ./send.py 10.0.2.2 “P4 is cool” 30 on h1, and iperf -c 10.0.2.22 -t 15 -u on h11.

We can now see the path the traffic takes from the switches.

How is it different from Traditional/OpenFlow Networking concepts?

This was implemented in the SDN final when doing a ping trace and finding which switches had packet increases from each switch and calculating the path from there. Here we can IDs to the packets and create a more detailed trace of the path taken.

### P4\_Runtime

First fill in the code TODOs

Code:

We can now see the path the traffic takes from the switches.

How is it different from Traditional/OpenFlow Networking concepts?

### Load\_balance

First fill in the code TODOs

Code:

# 2) Tunnel Transit Rule

    table\_entry = p4info\_helper.buildTableEntry(

        table\_name="MyIngress.myTunnel\_exact",

        match\_fields={"hdr.myTunnel.dst\_id": tunnel\_id},

        action\_name="MyIngress.myTunnel\_forward",

        action\_params={"port": SWITCH\_TO\_SWITCH\_PORT},

    )

    ingress\_sw.WriteTableEntry(table\_entry)

    print("Installed transit tunnel rule on %s" % ingress\_sw.name)

A screen shot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

All we do is make and h1 ping h2. And in another shell, run the ./my controller.py. Here we are adding flows instead of adding them through the switch’s CLI.

How is it different from Traditional/OpenFlow Networking concepts?

This is different from the concepts because we are mapping traffic based on tunnel ID. This amount of specificity is better than SDN because we can match on any value and manipulate as we want.

### P4\_Runtime

First fill in the code TODOs

Code:

We can now see the path the traffic takes from the switches.

How is it different from Traditional/OpenFlow Networking concepts?

### Source\_routing

First fill in the code TODOs

Code:

    state parse\_ethernet {

        packet.extract(hdr.ethernet);

        transition select(hdr.ethernet.etherType) {

            TYPE\_SRCROUTING: parse\_srcRouting;

            default: accept;

        }

    }

    state parse\_srcRouting {

        packet.extract(hdr.srcRoutes.next);

        transition select(hdr.srcRoutes.last.bos) {

            1: parse\_ipv4;

            default: parse\_srcRouting;

        }

    }

action srcRoute\_nhop() {

        standard\_metadata.egress\_spec = (bit<9>)hdr.srcRoutes[0].port;

        hdr.srcRoutes.pop\_front(1);

    }

apply {

        if (hdr.srcRoutes[0].isValid()){

            if (hdr.srcRoutes[0].bos == 1){

                srcRoute\_finish();

            }

            srcRoute\_nhop();

            if (hdr.ipv4.isValid()){

                update\_ttl();

            }

        }else{

            drop();

        }

A computer screen shot of white text

AI-generated content may be incorrect.

Do ./receive.py on h2 and ./send.py 10.0.2.2.

How is it different from Traditional/OpenFlow Networking concepts?

This is different because instead of a central controller knowing about traffic and where it is, we have the hosts sending the traffic. This is unique because we can track traffic from the host and the host controls the interaction between itself and the switch.